CARLETON UNIVERSITY

SCHOOL OF COMPUTER SCIENCE

COMP4905 – HONOURS PROJECT

Formation Flight of UAVs Swarms in an Obstacle–Filled Three–Dimensional Environment

Author
Yannick ABOUEM

Supervisor
Prof. Mark LANTHIER

August 19, 2024

Abstract

Pattern formation for swarms of robots is a novel problem in the field of robotics with many useful use cases, especially for swarms of UAVs. In this report we present a solution we developed for this problem using a leader-follower approach, where the leader decides the position the followers needs to reach, and we pair it with the path planning algorithm Field D*, to allow a swarm of UAVs to navigate an obstacle course while maintaining a certain formation. We finally test our solution in two scenarios to demonstrate that our solution works, albeit with some issues pertaining to the precision of the UAV's movements and the leader-follower paradigm.

Acknowledgements

This project uses Webots (http://www.cyberbotics.com), an open-source mobile robot simulation software developed by Cyberbotics Ltd.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Cooperation among different individuals in a group is often necessary to solve complex problems that are otherwise harder or impossible to complete for a singular individual. This concept is true in the field of robotics where robotic swarms have become an increasingly popular area of study. Robotic swarms are usually composed of many small and limited individual robots (Oh et al. 2017) collaborating to achieve a shared goal. Due to the early stage of development of this field, there are not many currently existing applications, however there exists many research projects to test the capabilities of swarm algorithms (Schranz et al. 2020). Despite the lack of representation in the industry sector, multi-robot systems have become of interest recently due to their abilities to resist failures and damages, adaptability to new environments and low costs (Oh et al. 2017).

A specific problem in swarm robotics is the pattern formation problem, which consists of "getting a group of robots to form and stay in a specific formation, like a wedge or a chain, and maintaining that formation" (Sri and Suvarchala 2022). The robots in the swarm need to coordinate to maintain a specific shape in order to achieve the desired goal. In this process, different methods of controlling the swarm can be used, for example the election of a leader or the imposition of a certain set of behaviors for each member. There is also a distinction between centralized and decentralized pattern formation, where centralized pattern formation is performed when there exists a centralized unit that coordinates the individual robots (Oh et al. 2017).

1.1 Swarms of Unmanned Aerial Vehicles

A swarm of Unmanned Aerial Vehicles is a swarm of robots where each robot is capable of flight using a set of rotors. These swarms have multiple uses. They are commonly used in military applications, but they are also used for civilian applications, such as aerial surveys, disaster management, environment mapping, search and rescue, and for leisure (Tahir et al. 2019). These are applications that would benefit from pattern formation, as we can direct the swarm to take an optimal shape for its use case. For example, we can choose a shape to best cover an area while performing an aerial survey or environment mapping, or adopt a chain-like formation to establish communication

between two points (Schranz et al. 2020) which can be deployed at a site of a natural disaster to aid in search and rescue efforts.

1.2 Objectives

The objective of this project is to provide a distributed solution to the pattern formation problem, meaning build a system of multiple robots capable of positioning themselves in a specific shape and maintain this shape while navigating an environment populated by obstacles. This will be achieved using obstacle detection and avoidance techniques, as well as distributed system principles, such as leader election, and computer network principles. For this project we chose to use unmanned aerial vehicles to demonstrate the efficacy of the solution. Similar work being done includes (Chen et al. 2021), where a hierarchical approach is applied to formation control of fixed wing UAVs, by breaking up the swarm into several smaller groups each headed by a leader.

2 Methodology

In this chapter we will explore the method used in and considered for this project. Since this project contains multiple sub-problems we decided to divide this chapter into four sections.

2.1 Automated flight

The first consideration for this project is about the autonomous flight of the UAVs, meaning that there should not be any input from the user regarding the flight control. This problem can then be broken down into two components: the control algorithm that controls the ability of the UAV to fly and the navigation algorithm that controls the direction of flight.

2.1.1 Control Algorithm

Flight controllers can be subdivided into two categories: linear controllers and non-linear controllers. Linear controllers use linear methods to control flight, which includes conventional controllers such as PID, H_∞ controller, gain scheduling and Linear Quadratic Regulator. On the other hand, non-linear controller are derived from the original dynamic model of the UAVs and arose to overcome shortcomings of linear controllers. They are generally harder to implement than linear controllers. Some examples are Feedback linearization, backstepping, sliding mode control, adaptive control, and model predictive control. (Nguyen et al. 2020)

For this project we settled on linear controller due to their simplicity to implement, in particular we considered the PID controller and the LQR controller.

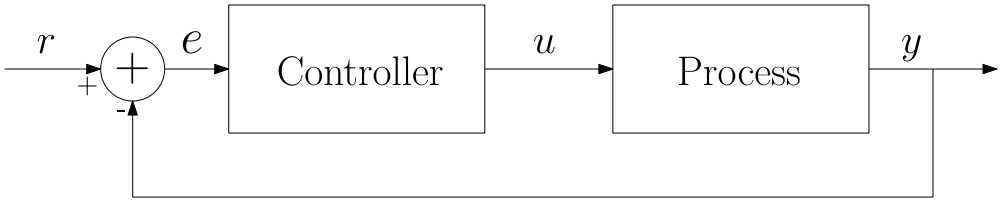


FIGURE 2.1: Diagram of a conventional PID controller

PID Control

The Proportional Integral Derivative controller is the most common controller used in industrial settings due to its usefulness and ease of implementation. (Lopez-Sanchez and Moreno-Valenzuela 2023) The controller is composed by a proportional element, an integral element and a derivative element, that represent the present, past and future error value(Araki 2009). It is usually designed in the structure represented in Fig. 2.1.

The controller first computes the error at a specific time t ($e(t)$) using Eq. (2.1).

$$e(t) = r(t) - y(t) \tag{2.1}$$

Where $r(t)$ is the set value and $y(t)$ is the real value of the process (Araki 2009). From the computed error the controller then finds a control factor $u(t)$ using a proportional coefficient (K_p), an integral coefficient (K_I) and a derivative coefficient (K_D) using Eq. (2.2) (Nguyen et al. 2020).

$$u(t) = K_p e(t) + K_I \int_0^t e(t)\mathrm{d}t + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{2.2}$$

In this project we decided to implement a PID controller due to the ease of implementation and the higher number of resources available at our disposal.

LQR Control

The other option considered, is the Linear-Quadratic Regulator controller (LQR controller), which is the optimal linear controller to operate a dynamic system at minimum cost (Nguyen et al. 2020).

$$J_{LQR} = \int_0^\infty y_i^T(t)Qy_i(t) + u_i^T(t)Ru_i(t)\mathrm{d}t \tag{2.3}$$

Eq. (2.3) computes the minimum quadratic cost (J_{LQR}) from the control input u_i and control output y_i, using two weight matrices (Q and R) (Nguyen et al. 2020).

2.1.2 Navigation Algorithm

For the navigation algorithm we used a simple algorithm where given a waypoint, we compute the derivative component of the PID algorithm. This can be achieved with the following algorithm.

Algorithm 1 Simple algorithm for UAV navigation using PID adapted from (Cyberbotics 2023) source code

Input: u, v, Y
Output: Y_d, P_d

$\alpha \leftarrow ((\text{atan2}(u_y - v_y, u_x - v_x) - Y) + 2\pi) \text{mod}(2\pi)$
if $\alpha > \pi$ **then**
    $\alpha \leftarrow \alpha - 2\pi$
end if
$Y_d \leftarrow K_{Yd_{max}} \frac{\alpha}{2\pi}$
$P_d \leftarrow \log(|\alpha|)$
if $P_d < K_{Pd_{max}}$ **then**
    $P_d \leftarrow K_{Pd_{max}}$
else if $P_d > 0.1$ **then**
    $P_d \leftarrow 0.1$
end if
return Y_d, P_d

Where u and v are vectors representing the UAV position and the waypoint position, Y is the current yaw angle of the UAV, Y_d and P_d are the yaw angle disturbance and the pitch angle disturbance respectively, and $K_{Yd_{max}}$ and $K_{Pd_{max}}$ are the maximum allowed disturbance of the yaw and pitch respectively.

Once the values of Y_d and P_d are computed they are then passed into the PID Controller which computes the amount of power to the engines necessary to move the UAV on the desired trajectory.

2.2 Obstacle Detection

There exist a wide array of sensors used in obstacle detections, some of these include passive sensors like cameras and infrared sensors, and active sensors like lidar, radar and ladar. Passive sensors absorb energy, like light or infrared waves, to create a picture of the environment while active sensors emit their own energy then collect the reflection emitted by the environment. Active sensors are better suited for our application, as they do not depend on atmospheric conditions or lighting conditions to operate. However, active sensors are prone to interference from other sources of electromagnetic waves. (Discant et al. 2007) For this project we decided to equip the UAVs with radar to detect obstacles.

Webots offers a simulation of a radar sensor which detects targets in the environment and provides their distance from the sensor, the power received back from the target, the speed relative to the sensor, and the horizontal angle (the azimuth) between the target and the sensor (Webots 2024). However, simply knowing this information about the obstacle is not sufficient as we need to find its position in order to avoid it.

2.2.1 Target Coordinates Extraction

To compute the position of a target we simply need its distance from the sensor, d, and the azimuth, α from the values returned from Webots API. Assuming that the UAV position is located at the origin, the distance between the UAV and the target can be considered a vector with magnitude d and the azimuth is the angle between the distance vector and the direction vector of the UAV. We call these vectors v and u respectively. Thus, what we are trying to find is $v = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$.

Given that we have two vectors and the angle between them, we can then find their dot product, where $\| v \| = d$, using Eq. (2.4).

$$u.v = \| u \| \| v \| \cos\alpha = u_x v_x + u_y v_y \tag{2.4}$$

We assume u is a unit vector. Since u is the direction of the UAV then the angle β is the angle between the vector u and the x-axis, and can be calculated from the heading of the UAV. Thus, we can compute the components of u using Eq. (2.5) and Eq. (2.6).

$$u_x = \cos\beta \tag{2.5}$$
$$u_y = \sin\beta \tag{2.6}$$

Therefore, we can rewrite Eq. (2.4) combined with the values of u_x and u_y resulting in Eq. (2.7).

$$\| v \| \cos \alpha = v_x \cos \beta + v_y \sin \beta \tag{2.7}$$

Similarly to u, we can also find the components of v using the angle between v and the x-axis, γ which is equal to $\beta - \alpha$. Using this angle we can write an equation to find the components of v.

$$\tan \gamma = \frac{v_y}{v_x} \tag{2.8}$$

Eq. (2.8) can then be written in terms of v_y resulting in Eq. (2.9).

$$v_y = v_x \tan \gamma \tag{2.9}$$

Finally, we can combine Eq. (2.7) with Eq. (2.9) to create one equation to find the x component of v.

$$v_x = \frac{d \cos \alpha}{\cos \beta + \tan \gamma \sin \beta} \tag{2.10}$$

Solving Eq. (2.10) and Eq. (2.9) will produce the coordinates of the target relative to the UAV. From there it is simply necessary to translate the vector v by the UAV position, resulting in the absolute position of the obstacle.

To avoid having the followers hit the obstacles we decided to add a buffer zone around each obstacle. These buffer zones have a traversal cost higher than the values for obstacle-free zones but lower than a zone with an obstacle. We do this by setting the traversal cost of each cell around the cell containing the obstacle to half the cost. So, each time an obstacle is detected, and its position computed, a specific cost c is assigned to the cell where the obstacle is located, and the neighboring cells cost is set to $\frac{c}{2}$.

2.3 Obstacle Avoidance

Once obstacles are detected, the UAVs should be capable of avoiding them and navigate between them until the swarm reaches the goal. As explained above, the navigation algorithm used in this project computes the trajectory to take to reach a specific waypoint,

so we can define the path the UAVs have to take as a list of waypoints. If we restrict the waypoints as being intersections of cells in a grid then we can abstract this grid as being a graph. Thus, what we wish to do, in this case, is to find a path from a starting node to a predefined end node, or search the graph for a specific node.

There exist many algorithms capable of solving this problem, with the most popular being A*. In the following subsections we will explore different algorithms considered for this project.

2.3.1 A*

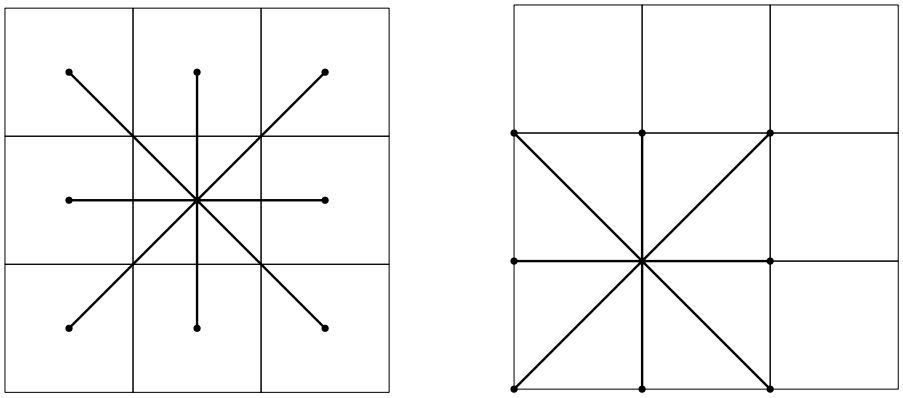
The most obvious and popular solution is the A* algorithm, which improves on Dijikstra's algorithm by using heuristics to decide which node should be considered next (*A* Search Algorithm* n.d.). However, since we are working in an unknown environment where the UAVs gain more knowledge of the obstacles locations as they progress, the path needs to be re-planned each time a new obstacle is found. This can become computationally expensive as we need to execute A* multiple times.

2.3.2 Path Planning in Partially-Known Environments with D*

In conditions where the UAVs do not have complete information about their surroundings it is advantageous to use an algorithm that assumes the possibility of potential changes in costs of the nodes. One of such algorithms is D*.

D* is capable of generating optimal paths for sensor equipped robots with a map of the environment that can be complete, partial or empty. The algorithm formulates the path planning process as a set of states, that represent potential robot locations, connected by directional arcs each with an associated cost. The algorithm maintains a list of states which cost has changed over time, and each time a cost is changed the affected states are added onto the list. Instead of re-planning the whole path, D* adjusts the path from the states where a change in cost happened and subsequent states. Thus, it operates faster in real-life scenarios than an algorithm that re-plans the whole trajectory each time a new obstacle is found. (Stenz 1994)

Despite its advantage over other algorithms, it is still possible to improve over D*. For example, D* Lite claims to be as efficient if not better than D* but with a simpler implementation (Koenig and Likhachev 2005).



(a) Representation used by standard algorithms (b) Representation used by Field D*

FIGURE 2.2: Representation of nodes positioning in standard path planning algorithms and field D*

2.3.3 Field D*: Improving D* Lite Using Interpolation

For this project we decided to implement Field D*, due to its promises of minimizing unnecessary turns and generating less expensive paths than D* and D* Lite. Field D* achieves this by using interpolation in the computation of the cost of the path. (Ferguson and Stentz 2006)

Most path planning algorithms that use a grid as underlying graph compute the cost of a node s, $g(s)$, using Eq. (2.11).

$$g(s) = \min_{s' \in nbrs(s)} [c(s, s') + g(s')] \tag{2.11}$$

Where $nbrs(s)$ is the set of neighbors of node s, $c(s, s')$ is the cost of traversing the edge between s and s', and $g(s')$ is the path cost of s'. This assumes that the only possible path is a straight line between the nodes, resulting in limitations as described by (Ferguson and Stentz 2006). In order to overcome these limitations, Field D* finds an approximation of the cost of the path to any cell boundary node s_b using linear interpolation. To do this, the algorithm assumes that the nodes of the graph are not located at the center of the cells as shown in Fig. 2.2a but rather at the vertices as shown in Fig. 2.2b (Ferguson and Stentz 2006). It then follows Algorithm 2 to compute the path cost, v_s, of a node s given two of its neighbors s_a and s_b.

Field D* is not a faster algorithm than D* Lite, with a planning time 1.7 times slower than its non-interpolation-based counterpart, but it offers a more economical path, 96%

as costly as D* Lite on average (Ferguson and Stentz 2006). Its path generation efficiency makes this algorithm competitive in some settings where minimizing the path cost is the main goal, for example in situation where the cost factor is the amount of fuel required to traverse an area.

Algorithm 2 Field D* path cost computation procedure by (Ferguson and Stentz 2006)

Input: s, s_a, s_b
Output: v_s

if s_a is diagonal neighbor of s **then**
  $s_1 \leftarrow s_b$
  $s_2 \leftarrow s_a$
else
  $s_1 \leftarrow s_a$
  $s_2 \leftarrow s_b$
end if
c is traversal cost of cell with corners s, s_1, s_2
b is traversal cost of cell with corners s, s_1 but not s_2.
if $\min(c, b) = \infty$ **then**
  $v_s \leftarrow \infty$
else if $g(s_1) \leq g(s_2)$ **then**
  $v_s \leftarrow \min(c, b) + g(s_1)$
else
  $f \leftarrow g(s_1) - g(s_2)$
  if $f \leq b$ **then**
    if $c \leq f$ **then**
      $v_s \leftarrow c\sqrt{2} + g(s_2)$
    else
      $y \leftarrow \min\left(\frac{f}{\sqrt{c^s - f^s}}, 1\right)$
      $v_s \leftarrow c\sqrt{1 + y^2} + f(1 - y) + g(s_2)$
    end if
  else
    if $c \leq b$ **then**
      $v_s \leftarrow c\sqrt{2} + g(s_2)$
    else
      $x \leftarrow 1 - \min\left(\frac{b}{\sqrt{c^2 - b^2}}, 1\right)$
      $v_s \leftarrow c\sqrt{1 + (1 - x)^2} + bx + g(s_2)$
    end if
  end if
end if
return v_s

2.3.4 Map of the Environment

In order for Field D* to function it requires access to a *"map"* of the environment where the cost of each cell and states are stored. This map is, essentially, a grid shaped graph, where each vertex is connected to eight neighbors, four adjacent and four diagonal. The node and its four adjacent neighbors form a cell, which has a specific traversal cost which is greater than zero. The cost to traverse an edge from a vertex to one of its adjacent neighbors are equal to the lowest traversal cost of the cells around the edge. While to traverse an edge from a vertex to a diagonal neighbor, the cost is $c\sqrt{2}$, where c is the traversal cost of the cell the edge lies into, as shown in (Ferguson and Stentz 2006). The map is used to compute the cost required to reach the goal, and it does not represent the movements the UAV can execute.

Algorithm 3 Field D* main loop

Input: M, s_{start}, s_{goal}, $OPEN$, $C_{changed}$
Output: W

for $x \in C_{changed}$ **do**
  for $s \in x$ **do**
    UPDATENODE(s')
  end for
end for
while $\min_{s \in OPEN}(key(s)) < key(s_{start})$ **or** $rhs(s_{start}) = g(s_{start}))$ **do**
  $s \leftarrow OPEN.pop()$
  $nbrs(s) \leftarrow M.getNbrs(s)$
  if $g(s) > rhs(s)$ **then**
    $g(s) \leftarrow rhs(s)$
    for $\forall s' \in nbrs(s)$ **do**
      UPDATENODE(s')
    end for
  else
    $g(s) \leftarrow \infty$
    for $\forall s' \in nbrs(s) \cup \{s\}$ **do**
      UPDATENODE(s')
    end for
  end if
end while
$W \leftarrow$ ExtractPath(M, s_{start}, s_{goal})
return W

In our project we decided to encode the map as a two–dimensional array, where each element is a cell containing four states. Using pointers we made such that each cell can share its states with neighboring cells, so we can avoid duplicate states. To extract states and cells from the map we developed query functions and other utility functions.

At the beginning of the program, the map is created, and its values are set to a predetermined default. All traversal costs are set to one and all states are given default values specified by Field D*, meaning that for each state s the values $g(s)$, which is the path cost of that state, and $rhs(s)$ which is the one-step lookahead path cost of s. We, then, identify the starting state (s_{start}) and the goal state (s_{goal}), from the initial GPS position of the UAV and a predefined set of coordinates respectively. We further initialize Filed D* by setting the value of $rhs(s_{goal})$ to zero, create a priority queue named $OPEN$, and we insert s_{goal} into the queue $OPEN$. Finally, we can compute the shortest path using Field D*, as shown in Algorithm 3, where M is the map, $C_{changed}$ is the set of cells whose traversal cost changed and W is the set of coordinates that are points on the shortest path in order.

Algorithm 4 UpdateNode function

Input: $M, s_{goal}, OPEN$

```
1:  function UPDATENODE(s)
2:      if s was not visited before then
3:          g(s) ← ∞
4:      end if
5:      if s ≠ s_goal then
6:          connbrs(s) ← M.getConnbrs(s)
7:          for ∀(s', s'') ∈ connbrs(s) do
8:              c ← COMPUTECOST(s, s', s'')
9:              if c < rhs(s) then
10:                 rhs(s) ← c
11:             end if
12:         end for
13:     end if
14:     if s ∈ OPEN then
15:         OPEN.remove(s)
16:     end if
17:     if g(s) ≠ rhs(s) then
18:         OPEN.insert(s, key(s))
19:     end if
20: end function
```

2.3.5 Path Extraction

The last step in the obstacle avoidance is to extract the path the UAVs will fly from the map. Since we are using Field D*, to extract the path we begin at the start node and iteratively compute the boundary point to move to next (Ferguson and Stentz 2006). We compute such point depending on the g-value of the two consecutive neighbors of the current node, s. Given $f = g(s_1) - g(s_2)$, where s_1 and s_2 are consecutive neighbors

of s, we have three different cases as shown in (Ferguson and Stentz 2006):

1. If $f \leq 0$ then s_1 is the least expensive boundary point to move to next.

2. If $f > b$, where b is the cost of the cell with vertices nodes s ans s_1, then the least expensive path involves moving for a certain distance x and then moving straight to node s_2. The value of x can be found by interpolation using Eq. (2.12) using the cost of the cell containing s, s_1 and s_2, as c.

$$x = 1 - \min\left(\frac{b}{c^2 - b^2}, 1\right) \tag{2.12}$$

3. Otherwise, the least expensive path is a direct path to some point s_y located on the $s_1 s_2$ edge at a certain distance y from s_1. Similarly to the previous case, the value of y can be found through interpolation using Eq. (2.13).

$$y = \min\left(\frac{f}{c^2 - f^2}, 1\right) \tag{2.13}$$

Due to limitations in the UAV precision in its movements, we decided to alter case 2 slightly by not computing the distance x and simply directing the UAV to fly straight to s_2 through the cell.

Once a waypoint has been identified we continue the computation by choosing the node with lowest g-value between s_1 and s_2, and iterate until the current state is equal to the goal state. Algorithm 5 is the complete path extraction procedure used in the project.

Algorithm 5 Path extraction algorithm for Field D*

Input: M,s_{start},s_{goal}
Output: W

function EXTRACTPATH(M,s_{start},s_{goal})
  $s \leftarrow s_{start}$
  while $s \neq s_{goal}$ **do**
    $(c, d) \leftarrow M.getCellsFromStates(s, s_1(s), s_2(s))$
    $f \leftarrow g(s_1(s)) - g(s_2(s))$
    if $f \leq 0$ **then**
      $W.add(s_1(s))$
    else if $f > b$ **then**
      $W.add(s_2(s))$
    else
      $y \leftarrow \min\left(\frac{f}{c^2 - f^2}, 1\right)$
      $s_y \leftarrow s_1(s) + y$
      $W.add(s_y)$
    end if
    if $g(s_1(s)) < g(s_2(s))$ **then**
      $s \leftarrow s_1(s)$
    else
      $s \leftarrow s_2(s)$
    end if
  end while
  return W
end function

2.4 Pattern Formation

The central aspect of this project is pattern formation, which we already discussed in the previous chapter. In this section we will discuss the technical detail behind our solution to the pattern formation project.

2.4.1 Leader-follower structure

In this project, we organized the UAVs in a structure composed of one leader and multiple followers. The goal of the leader is to plan a path to reach the goal and communicate each way points to the followers, while the followers simply fly towards the waypoints assigned by the leader. Each waypoint is computed in order to preserve a specific shape dictated by the initial position of the UAVs.

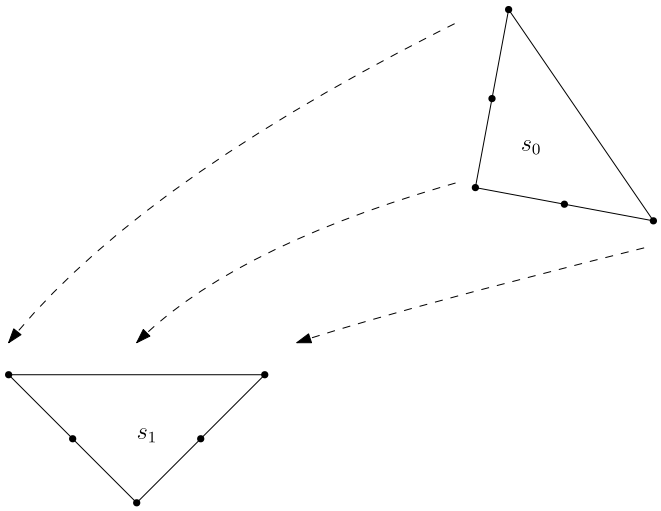


FIGURE 2.3: Graphic of desired pattern transposition from one waypoint to another

To achieve this each waypoint for each follower is computed not only from the distance from the leader but also depending on the expected heading the leader will have. The result is expected to look similar to Fig. 2.3 where the structure of the formation is not changed when a turn occurs.

Let θ be the angle between the leader UAV direction and the x-axis, we can use this angle as rotation angle for the formation. We cannot find the precise heading of the leader UAV due to possible variance in the real path the UAVs will take, but we can approximate this value given two waypoints. Let v and w be the current and next waypoints for the leader UAV. If we compute the difference of these two waypoints we

Table 2.1: Approximations of θ based on difference of $w - v$

Δx	Δy	Approx. θ
$+$	0	0
0	$+$	$\frac{1}{2}\pi$
$-$	0	π
0	$-$	$\frac{3}{2}\pi$
$+$	$+$	$\frac{1}{4}\pi$
$-$	$+$	$\frac{3}{4}\pi$
$-$	$-$	$\frac{5}{4}\pi$
$+$	$-$	$-\frac{1}{4}\pi$

can find what direction the UAV is facing when reaching this waypoint. We show these approximations in Tab. 2.1, where $+$ is a positive difference or $w_x > v_x$, $-$ is a negative difference or $w_x < v_x$ and 0 is no difference or $w_x = v_x$[1].

Once we found the value of θ we transform the orientation of the formation using a rotation matrix as shown in Eq. (2.14).

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{2.14}$$

Once the formation is rotated it is then translated with respect to the leader's waypoint.

However, this solution works only if the robots are restricted to specific headings, which we found that this is not the case in our project. The use of interpolation in path planning means that θ can be of any value. Thus, to make the formation more accurate we decided to take a more proactive approach in computing the value of the angle. As pointed above, θ is the angle between the UAV direction and the x-axis, so it can be computed with ease. Let u and v be the current and next waypoint respectively, then $\hat{w} = \begin{bmatrix} v_x - u_x \\ v_y - u_y \end{bmatrix}$ is the distance vector between u and v and has magnitude $\| \hat{w} \| = \sqrt{(v_x - u_x)^2 + (v_y - u_y)^2}$. Using a vector parallel to the x-axis $\hat{x}$ we can compute the dot product of these two vectors using Eq. (2.15), which is arranged with respect of θ as in Eq. (2.16).

[1]Note: in this example we use the x components of v and w, but the principle is the same for the y components.

$$\hat{w}.\hat{x} = \parallel \hat{w} \parallel \parallel \hat{x} \parallel \cos\theta = \hat{x}_x\hat{w}_x + \hat{x}_y\hat{w}_y \tag{2.15}$$

$$\theta = \arccos\left(\frac{\hat{x}_x\hat{w}_x + \hat{x}_y\hat{w}_y}{\parallel \hat{w} \parallel \parallel \hat{x} \parallel}\right) \tag{2.16}$$

Replacing in the components of $\hat{x}$ and $\hat{w}$ we get Eq. (2.17).

$$\theta = \arccos\left(\frac{v_x - u_x}{\sqrt{(v_x - u_x)^2 + (v_y - u_y)^2}}\right) \tag{2.17}$$

After this step we simply rotate and translate using θ and u. This method is more advantageous than the one we explained above as it is not limited to a finite predefined amounts of headings.

3 Program Structure and Inter-UAV communication

In this chapter we will explore in more detail the program we designed to illustrate our solution to the pattern formation problem.

3.1 Program Structure

The program is composed by three phases: an initialization, where all variables and parameters are initialized, a main loop, where all movements and inter-UAV communication happens, and a final clean-up phase. Each UAV's phase is independent of other UAVs and dictated by a state variable that can assume one of the values presented in Table 3.1. The state also serves to determine whether a UAV has been elected leader or not during the execution of the program.

A full overview of the program structure is available in Fig. 3.1.

Table 3.1: Possible values of the state variable

State value	Description
`INIT`	The UAV is in the initialization phase
`RUN`	The UAV is in the main loop phase and has been elected leader
`END`	The goal was reached, and the UAV is in the clean-up phase
`F_RUN`	The UAV is in the main loop and has not been elected leader
`F_WAIT`	The UAV is waiting on instructions from the leader, and it is not the leader

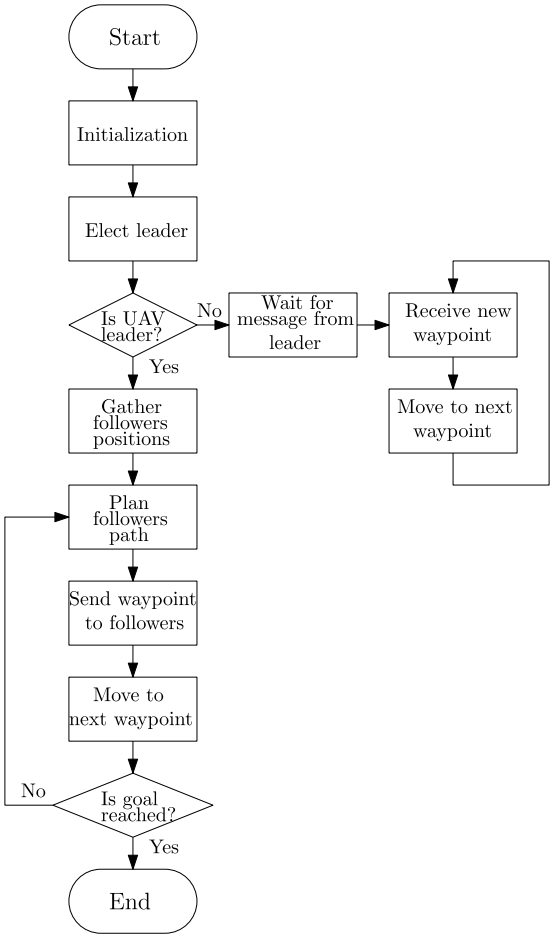


FIGURE 3.1: The flowchart diagram of the program

3.2 Inter-UAV communication

In order for the UAVs to cooperate together we equipped each UAV with a transmitter and a receiver to send messages to each other. The content of these messages varies depending on the phase of the program but are generally used to send either the UAV identifier, used for leader election and for message routing, their initial position, used by the leader to determine the shape of the formation, and the waypoint the followers need to fly to next. Each message is a packet composed of two parts: a header and message. The header contains metadata necessary to route the message to the correct UAV and the type of message, while the message part contains the content of the message which is usually a set of coordinates. A diagram of the packet is shown in Fig. 3.2.

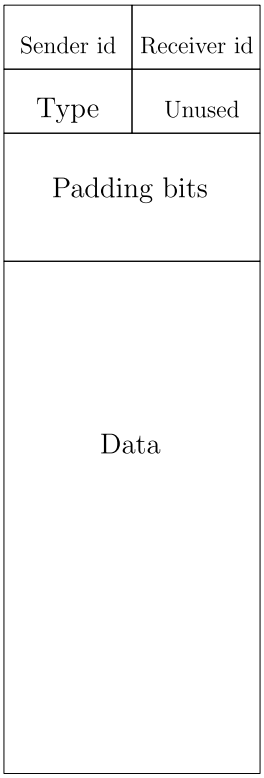


FIGURE 3.2: Diagram of a message. The diagram is 16 bits in width.

The header is 4 bytes in size, each field in the header occupy a space of 1 byte, including an unused byte. The data part of the message has a maximum size of 16 bytes, however it is only used in messages carrying information about the position of the UAV and waypoints. There are additional four bytes of padding between the header and the data.

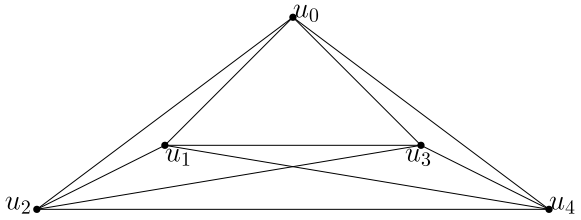


FIGURE 3.3: Network topology diagram

To keep the network simple and decrease the number of failures, all UAVs are connected to each other in a complete network as shown in Fig. 3.3.

3.2.1 Leader Election

To implement the pattern formation we need one of the UAVs to become a leader and plan the path. To achieve this we decided to implement the FloodMax algorithm for leader election. This is a simple algorithm that allows leader election in any network as long as the diameter of the network, D, is known, achieving this in a number of rounds equal to the diameter of the network (Kranakis 2023). This is particularly effective in our case, as the diameter of a complete network is 1. See Algorithm 6 for the steps of FloodMax.

Algorithm 6 FloodMax Algorithm

Input: D

1: Each node maintains a record of the highest ID it has seen so far
2: At each round the node propagates this maximum to all neighboring nodes
3: After D rounds if the maximum recorded is equal to the node ID the node elects itself as leader

4 Results

In this section we will showcase the result obtained by applying the concept discussed above as well as some test scenarios we designed to demonstrate the effectiveness of the solution presented.

4.1 Test scenario

We constructed two tests scenarios to test the effectiveness of the solution presented. The first scenario is intended to verify the correctness of the obstacle avoidance component of this project, while the second scenario was designed to test the pattern formation component. Both scenarios use a similar obstacle course with the only difference in the number of obstacles and the number of UAVs used, being one in the first and three in the second. Fig. 4.1 and Fig. 4.2 show a map of the obstacle courses used in the first scenario and the second scenario respectively, where u is the start position of the UAVs and v is the goal the UAVs need to reach.

The reason why we decided to design two scenarios to test the solution is due to the UAVs having difficulty avoiding obstacles while maintaining a formation. This was caused by a lack of space available for the swarm to pass through, thus we designed a second scenario with fewer obstacles for the formation only.

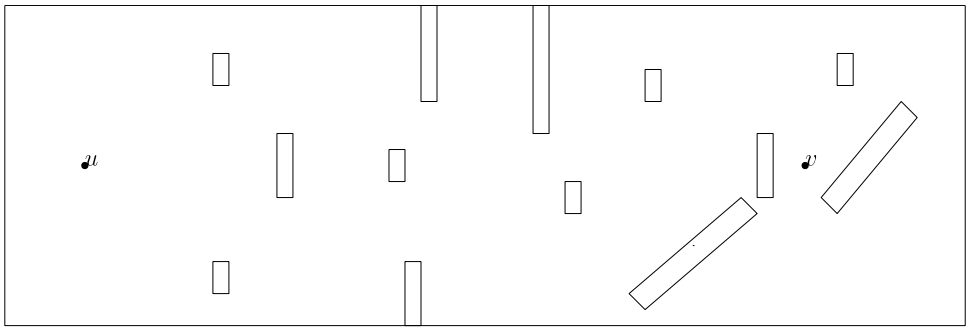


FIGURE 4.1: The map of the obstacle course for the first scenario

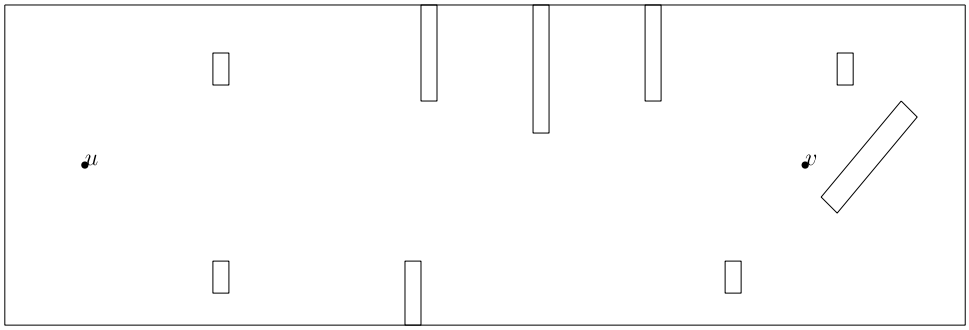


FIGURE 4.2: The map of the obstacle course for the second scenario

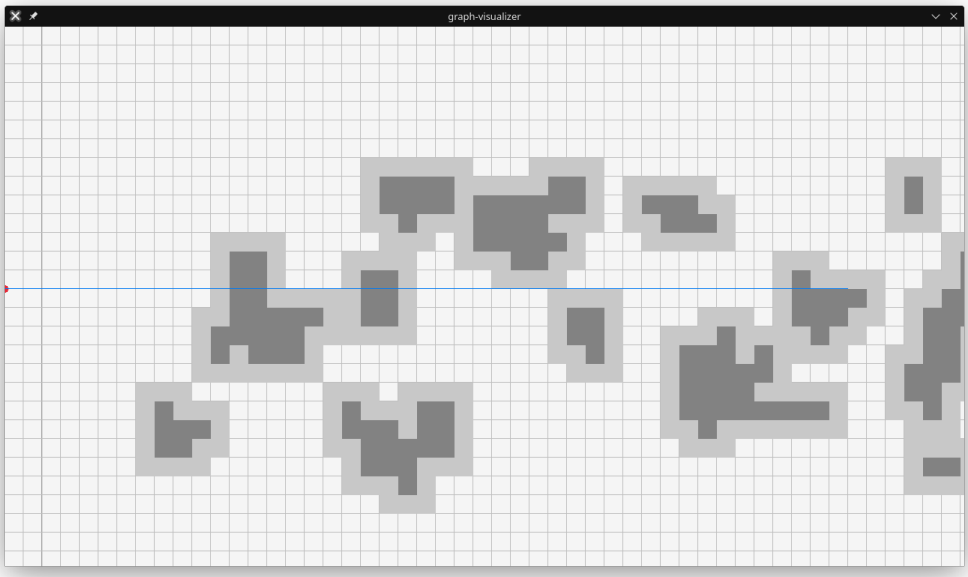
4.2 Observations from scenarios

In this section we will discuss observations and results obtained from the test scenarios above. Overall, in both scenarios the UAVs were able to carry out the defined tasks. However, the final result is far from perfect. During the execution multiple incorrect behavior were observed, including some UAVs travelling at a slower speed than other UAVs in the second scenario and the UAV not following the shortest path, as determined by Field D*, in the first scenario. In the following subsections we will have a closer look at these issues.

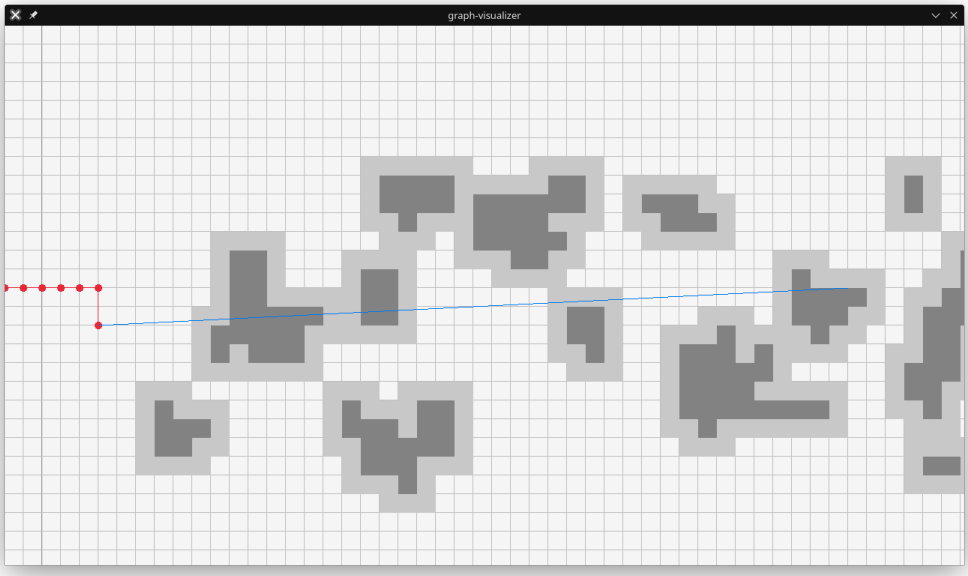
4.2.1 First Scenario

As explained previously, the first scenario is intended to test the obstacle avoidance of the presented solution. It involves a single UAV and the obstacle course shown in Fig. 4.1. Due to the difficulty in observing the path the UAV has to take in order to reach the goal, we developed a companion program to visualize this path from data extracted from the Webots simulation. In the images captured from the execution of this program the red dots are the waypoints the UAV has to reach, the red line represents the path between each waypoint, and the blue line is the shortest distance between the last waypoint we reached and the goal, while the dark gray squares are areas with high cost, meaning there is an obstacle in that area, and the light gray squares are areas with lower cost than the darker squares.

As seen in Fig. 4.3, Fig. 4.3, Fig. 4.3, and Fig. 4.3, the path computed by our project allows the UAV to avoid obstacles while minimizing the cost of this path. In practice, however the UAV does not always follow this path strictly, but it deviates due to physical limitations of the UAV. For example, if we compare the UAV motion at the beginning of the simulation with the path shown in Fig. 4.3b, we see that the UAV performs some

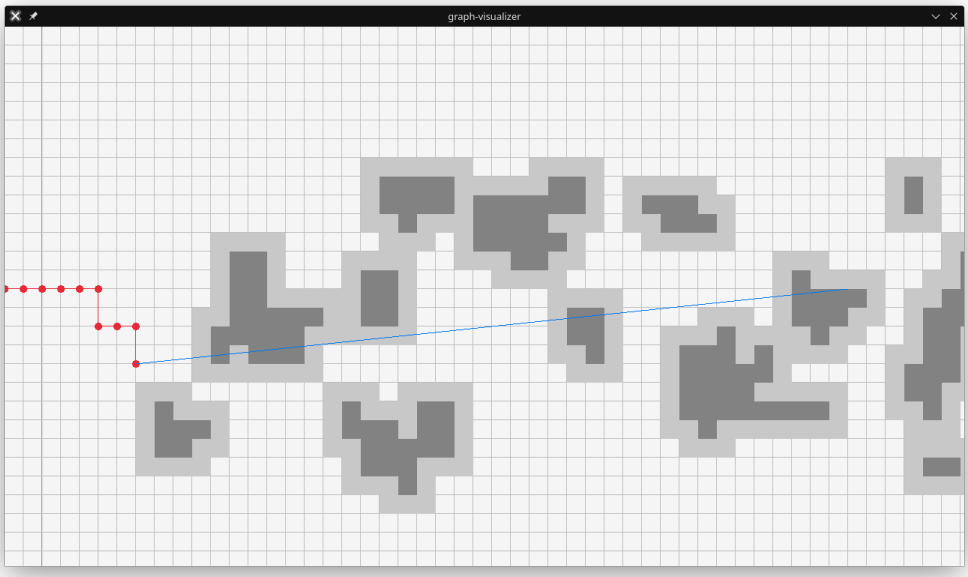


(a) At the beginning of the program the UAV has no data on the obstacles, so it assumes the shortest path must be a straight line to the goal

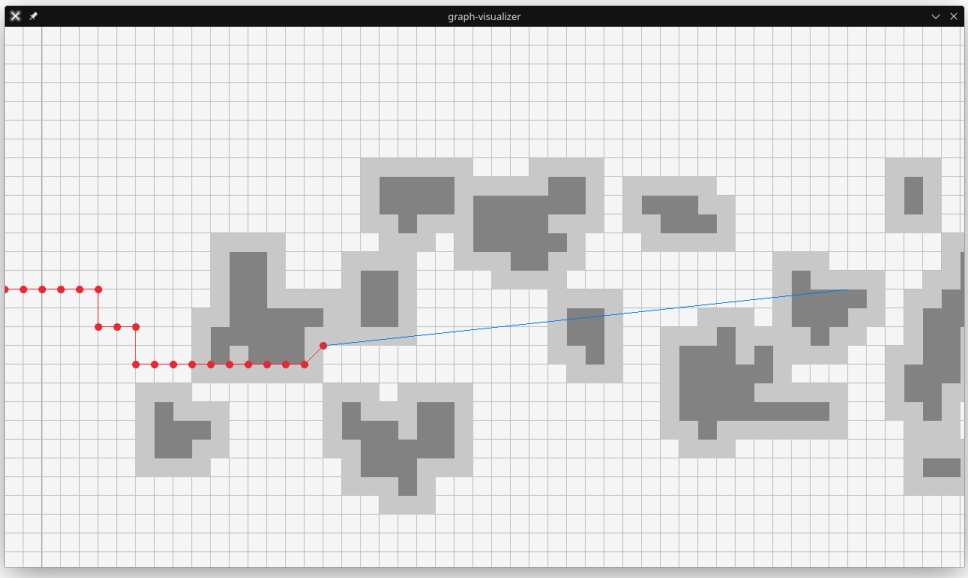


(b) As the program continues it detects an obstacle in front of it, and it turns to avoid it

FIGURE 4.3: Path computed by Field D* - Part 1

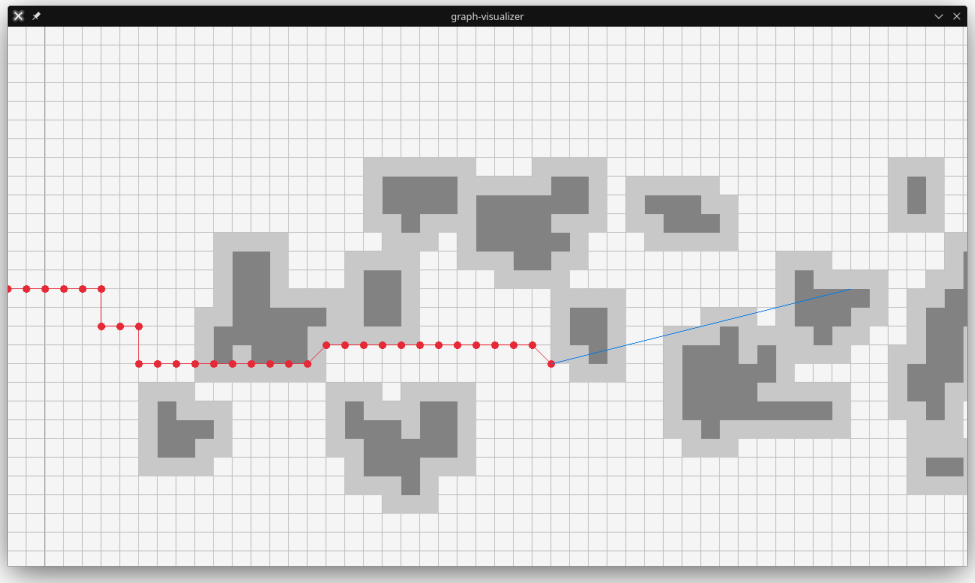


(a) It turns until it is clear of the obstacle

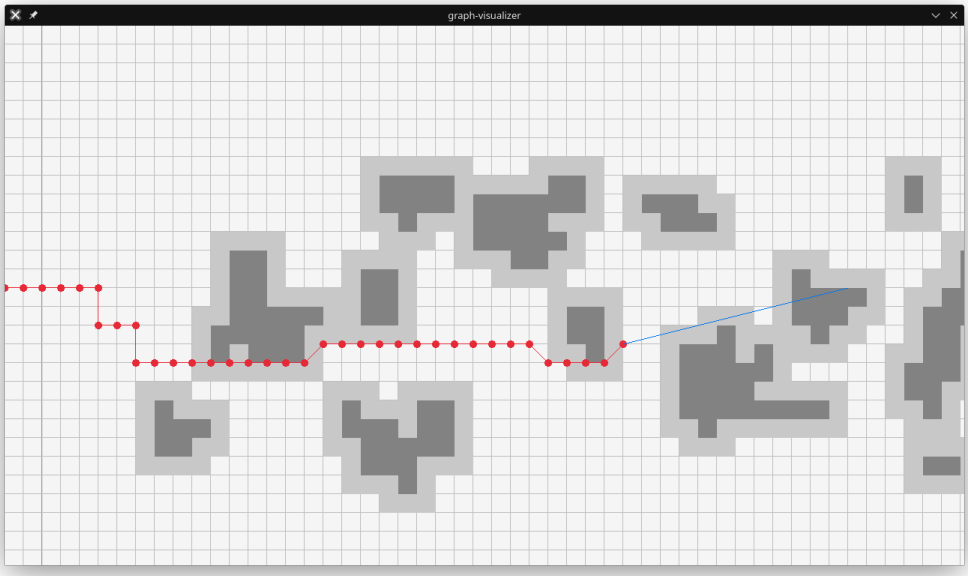


(b) After avoiding the obstacle it attempts to return on the shortest path between the start and the goal

FIGURE 4.4: Path computed by Field D* - Part 2

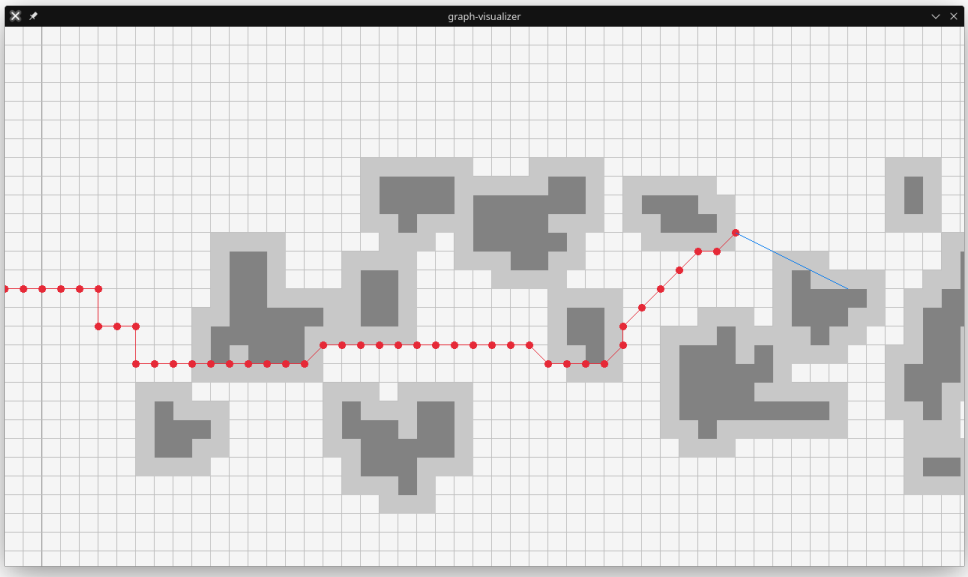


(a) It is unable to return on the shortest path, as there is an obstacle in the way, so it turns towards the goal and continues in a straight line until a new obstacle is found

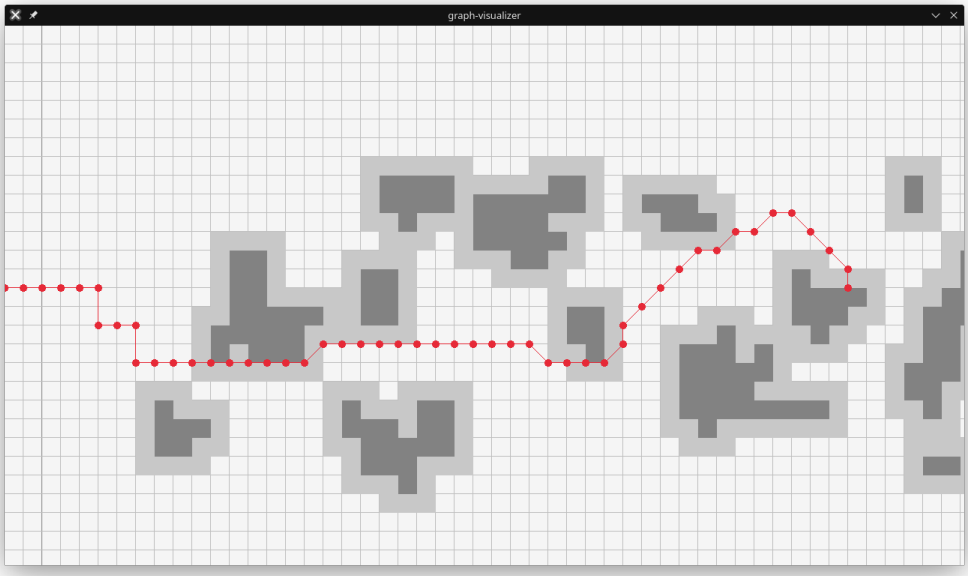


(b) Avoids another obstacle and detects an obstacle in front of it

FIGURE 4.5: Path computed by Field D* - Part 3



(a) It continues upwards to avoid a larger obstacle and diagonally to avoid one in front of it



(b) The UAV turns around to reach the goal

FIGURE 4.6: Path computed by Field D* - Part 4

turns that are not in the computed path, that's because the UAV drifts from the path and needs to adjust in order to return on the path. Another case is after 4 minutes and 2 seconds from the beginning of the simulation, where the UAV does a 180-degree turn because it has overshot one of the waypoints and needs to turn back, instead of turning upwards and then forwards as shown in 4.5a.

4.2.2 Second scenario

The second scenario is aimed at testing the formation capabilities of the project. It is a simple scenario where three UAVs in a V-shaped formation fly while avoiding few obstacles as shown in Fig. 4.2. The UAVs were able to reach the goal without touching any of the obstacles and roughly maintaining the predefined formation. However, there were a few situations in which the shape of the formation was violated. One such cases is at 1 minute into the simulation where the UAVs break the formation in order to reposition after a turn to the left. This movement causes the UAV at the bottom of the formation to fly at a slower speed and trail behind the other UAVs. This is because the UAV turns excessively, and thus it has to adjust its heading before moving forward. The issue can be seen better after 2 minutes and 30 seconds from the start of the program. We can see the lower UAV turning slightly to its left and then turn right to face forward, while the other UAV had already begun moving at that point, leaving the UAV behind. Thar slight left turn is caused by the leader sending the new waypoint to the UAV later than expected, causing the UAV to retain the older waypoint and deviate from the path. Fig. 4.7 shows the problematic behavior.

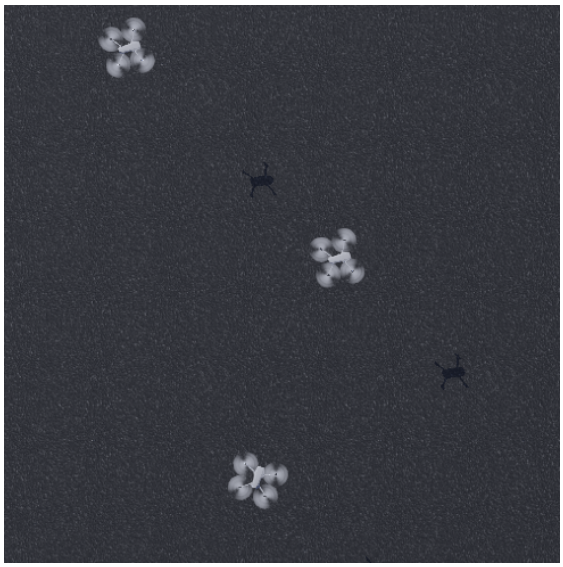


FIGURE 4.7: Image of the bottom UAV turning excessively to the left

5 Conclusion

In this project we explored a solution to pattern formation of robotic swarms, a novel research topic in the field of robotics. Our presented solution makes use of a swarm Unmanned Aerial Vehicles, controlled by a PID controller, which needs to navigate through an obstacle course while maintaining a specific formation. To achieve this we implemented the path planning algorithm Field D* that computes the lowest cost path from the start point to the goal point. We paired this algorithm alongside a radar to detect the position of the obstacles and used a custom algorithm to extract the obstacles position.

In order to create the formation we adopted a leader-follower structure where the leader finds the path the formation has to fly and communicates the path the followers have to take. We compute the followers paths from the distance the UAVs have at the beginning of the simulation and the angle between two waypoints in the path. Moreover, we use distributed computing concept, such as leader election.

We tested our solution in two scenarios to test the obstacle avoidance part and formation part. The first scenario was completed successfully by the UAV, however we witnessed some issues caused by the poor precision of the UAV controller. The second scenario was also completed successfully by the swarm, but the formation shape was not always maintained correctly because of excessive turning by one UAV.

Bibliography

A Search Algorithm* (n.d.). Webpage. Ada Computer Science. URL: https://adacomputerscience.org/concepts/path_a_star?examBoard=ada&stage=all (visited on 07/28/2024).

Araki, M. (2009). "PID Control". In: *Control Systems, Robotics, and Automation*. Ed. by Heinz Unbehauen. 2nd ed. UNESCO-EOLSS, pp. 58–65.

Chen, Hao et al. (2021). "Formation flight of fixed-wing UAV swarms: A group-based hierarchical approach". In: *Chinese Journal of Aeronautics* 34.

Cyberbotics (2023). *Maveric 2 Pro Patrol Controller*. source code. URL: https://raw.githubusercontent.com/cyberbotics/webots/master/projects/robots/dji/mavic/controllers/mavic2pro_patrol/mavic2pro_patrol.py.

Discant, Anca et al. (2007). "Sensors for Obstacle Detection - A Survey". In: *2007 30th International Spring Seminar on Electronics Technology (ISSE)*, pp. 100–105. DOI: 10.1109/ISSE.2007.4432828.

Ferguson, Dave and Anthony Stentz (2006). "Using Interpolation to Improve Path Planning: The Field D* Algorithm". In: *Journal of Field Robotics* 23.2, pp. 79–101. DOI: 10.1002/rob.20109.

Koenig, Sven and Maxim Likhachev (June 2005). "Fast Replanning for Navigation in Unknown Terrain". In: *IEEE Transaction On Robotics* 21.3, pp. 354–363.

Kranakis, Evangelos (Nov. 2023). *Personal Communication*.

Lopez-Sanchez, Ivan and Javier Moreno-Valenzuela (2023). "PID control of quadrotor UAVs: A survey". In: *Annual Reviews in Control* 56.

Nguyen, Hoa et al. (May 2020). "Control Algorithms for UAVs: A Comprehensive Survey". In: *EAI Endorsed Transactions on Industrial Networks and Intelligent Systems* 7, p. 164586. DOI: 10.4108/eai.18-5-2020.164586.

Oh, Hyondong et al. (2017). "Bio-inspired self-organising multi-robot pattern formation: A review". In: *Robotics And Autonomous Systems* 91.

Schranz, Melanie et al. (2020). "Swarm Robotic Behaviors and Current Applications". In: *Frontiers in Robotics and AI*. doi: 10.3389/frobt.2020.00036. PMID: 33501204; PMCID: PMC7805972.

Sri, S.M. Lakshmi and C.V. Kavya Suvarchala (2022). "Multi-robot systems: a review of pattern formation and adaptation". In: *Advanced Engineering Sciences* 54.

Stenz, Anthony (May 1994). "Optimal and Efficient Path Planning for Partially-Known Environments". In: *IEEE International Conference on Robotics and Automation*, IEEE.

Tahir, Anam et al. (2019). "Swarms of Unmanned Aerial Vehicles — A Survey". In: *Journal of Industrial Information Integration* 16.

Webots (2024). *http://www.cyberbotics.com/doc/reference/radar*. Ed. by Cyberbotics Ltd. Open-source Mobile Robot Simulation Software. URL: `http://www.cyberbotics.com/doc/reference/radar` (visited on 07/31/2024).